Experience with processes and monitors in Mesa
(summary)

Butler W. Lampson
Xerox Research Center
Palo Alto, California 94304

David D. Redell
Xerox System Development Department
Palo Alto, California 94304

In early 1977 we began to design the concurrent programming facilities of Pilot, a new operating system for a personal computer [5]. Pilot is a fairly large program itself (25,000 lines of Mesa code). In addition, it supports some large applications, ranging from data base management to internetwork message transmission, which are heavy users of concurrency (our experience with some of these applications is discussed in the paper). We intended the new facilities to be used at least for the following purposes:

Local concurrent programming: An individual application can be implemented as a tightly coupled group of synchronized processes to express the concurrency inherent in the application.

Global resource sharing: Independent applications can run together on the same machine, cooperatively sharing the resources; in particular, their processes can share the processor.

Replacing interrupts: A request for software attention to a device can be handled directly by waking up an appropriate process, without going through a separate interrupt mechanism (e.g., a forced branch, etc.).

Pilot is closely coupled to the Mesa language [4], which is used to write both Pilot itself and the applications programs it supports. Hence it was natural to design these facilities as part of Mesa; this makes them more convenient to use, and also allows the compiler to detect many kinds of errors in their use. The idea of integrating such facilities into a language was certainly not new. Furthermore, the invention of monitors by Dijkstra, Hoare and Brinch Hansen [1, 2, 3] provided a very attractive framework for reliable concurrent programming.

We therefore thought that our task would be an easy one: read the literature, compare the alternatives offered there, and pick the one most suitable for our needs. This expectation proved to be naive. Because of the large size and wide variety of our applications, we had to address a number of issues which were not dealt with in the published work on monitors. Notable among these are :

a) Program structure: Mesa has facilities for organizing programs into modules which communicate through well-defined interfaces. Processes must fit into this scheme .

b) Creating processes: a fixed number of processes is unacceptable in such a general-purpose system.

c) Creating monitors: a fixed number of monitors is also unacceptable, since the number of synchronizers should be a function of the amount of data.

d) A WAIT in a nested monitor call: this issue had been (and has continued to be) the source of a considerable amount of confusion.

e) Exceptions: a realistic system must have timeouts, and it must have a way to abort a process.

f) Scheduling: the precise semantics of waiting on a condition variable was not agreed upon.

g) Input-output: the details of fitting devices into the framework of monitors and condition variables had not been worked out.

Processes

Mesa allows any procedure to be invoked in a new process, which continues execution independently; later the results can be retrieved. This pattern of process creation allows a simple procedure call

n ← *ReadLine*[*terminal*]

to be done concurrently:

p ← FORK *ReadLine*[*terminal*];
. . . <concurrent computation> . . .
n ← JOIN *p* ;

The important properties of this scheme are that

It treats a process as a first-class value in the language.

The method for passing parameters is exactly the same as for procedures, and is subject to the same strict type checking.

No special declaration is needed for a procedure which is invoked as a process.

The cost of creating and destroying a process is only a few times the cost of calling a procedure.

Monitors

When several processes interact by sharing data, care must be taken to properly synchronize access to the data. The idea behind monitors is that a proper vehicle for this interaction is one which unifies

the synchronization,
the shared data,
the body of code which performs the accesses.

The data is *protected* by a *monitor*, and can only be accessed within the body of a *monitor procedure*. The processes can only perform operations on the data by calling monitor procedures. The monitor ensures that at most one process is executing a monitor procedure at a time; this process is said to be *in* the monitor. As long as any order of calling the entry procedures produces meaningful results, no additional synchronization is needed

In Mesa the simplest monitor is an instance of a *module*, which is the basic unit of global program structuring, used in sequential programming to implement a data abstraction. Such a module has PUBLIC procedures which constitute the external interface to the abstraction, and PRIVATE procedures which are internal. In a MONITOR module the PUBLIC procedures acquire and release the monitor lock; PRIVATE ones do not, since they can only be called from PUBLIC procedures or other internal procedures.

Within a monitor a process can WAIT on a *condition variable* until some other process is in the monitor and does a NOTIFY to the variable. The WAIT releases the monitor lock, which is reacquired when the waiting process reenters the monitor. If, however, the monitor calls some procedure outside the monitor module, the lock is not released, even if the other procedure is in (or calls) another monitor and ends up doing a WAIT. Otherwise, the invariant maintained by the monitor would have to be established before every such call. The result would be to make calling such procedures hopelessly cumbersome. Of course, holding locks longer increases the chances of deadlock. We have seen two patterns of deadlock, which are discussed in the paper.

When an exceptional condition causes part of a computation to be abandoned, each procedure being abandoned is given a chance to clean up. This feature of Mesa interacts with concurrency: if an entry procedure is abandoned, the invariant must first be restored (the programmer's job) and the monitor lock released (done automatically).

Condition variables

Hoare's definition of monitors [3] requires that a process waiting on a condition variable must run immediately when another process signals that variable, and that the signalling process in turn runs as soon as the waiter leaves the monitor. This definition allows the waiter to assume the truth of some predicate stronger than the monitor invariant (which the signaller must of course establish), but it requires several additional process switches whenever a process continues after a wait.

Mesa takes a different view: a NOTIFY is regarded as a *hint* to a waiting process; there is no guarantee that some other process will not enter the monitor first. Hence only the monitor invariant may be assumed after a wait. The proper pattern of code for a wait is therefore:

WHILE NOT <OK to proceed> DO WAIT *c* ENDLOOP.

The verification rule for Mesa monitors is thus extremely simple: the monitor invariant must be established just before a RETURN from an entry procedure or a WAIT, and it may be assumed at the start of an entry procedure and just after a WAIT.

With this rule it is easy to add three additional ways to resume a waiting process:

Timeout: A process which has been waiting for a preset time *t* will resume automatically. Presumably in most cases it will check the time and take some recovery action before waiting again.

Abort: Any process may be aborted The effect is simply that the process resumes at once from the next WAIT, and the *Aborted* exception occurs. This mechanism allows one process to gently prod another.

Broadcast: this causes *all* the processes waiting on a condition to resume; NOTIFY is an optimizzation.

None of these mechanisms affects the proof rule for monitors at all. Each provides a way to attract the attention of a waiting process at an appropriate time.

Communication with input/output devices is also handled by monitors and conditions. A process can NOTIFY a special kind of condition variable to attract the attention of a device. Reciprocally, a device can NOTIFY a condition variable to resume a waiting process

Applications

The full paper discusses three substantial applications of these facilities: the Pilot system itself, a distributed replicated file system, and an internetwork gateway.

References

1. Brinch Hansen, P., *Operating System Principles*, Prentice-Hall, 1973.
2. Brinch Hansen, P. "The programming language Concurrent Pascal," *IEEE Transactions on Software Engineering* **1**, 2, pp 199-207 (June 1975).
3. Hoare, C.A.R., "Monitors: An operating system structuring concept," *Comm. ACM* **17**, 10, pp 549-557, (Oct 1974).
4. Mitchell, J.G., Maybury, W. and Sweet, R., *Mesa Language Manual*, Report CSL-79-3, Xerox Research Center, Palo Alto, CA, 1979.
5. Redell, D. et. al., "Pilot: An operating system kernel for a personal computer," to appear in *Comm. ACM*.